Notions de type et de fonction

Doc. 2,1

Objectif

En manipulant Python dans la console et dans Thonny, vous allez chercher à répondre aux questions suivantes :

- Comment un programme **représente-t-il** une information (un âge, un nom, une réponse par oui ou non) ?
- Comment Python **calcule-t-il** le résultat d'une expression, et dans quel ordre ?
- Que se passe-t-il quand on **appelle** une fonction, et comment savoir s'en servir ?
- Comment **fabriquer** sa propre fonction et vérifier qu'elle fait ce qu'on attend d'elle ?

§1. Comment représenter une information dans un programme ?

L'informatique consiste à automatiser le traitement d'informations. Pour effectuer ces traitements, un programme doit d'abord être capable de **représenter** les informations issues du monde réel.

Un âge, une température, un nom, une réponse à une question ou une distance ne sont pas des informations de même nature. Un langage de programmation utilise donc différents **types** pour **modéliser** ces différentes informations.

Q1. Pour chacune des informations suivantes, proposer une manière simple de la représenter à l'aide de Python :

- le nombre d'élèves d'une classe ;
- une température de 18,5 °C ;
- le nom d'un élève ;
- la réponse à la question « l'élève est-il majeur ? ».

Tester ensuite dans la console :

```Python
>>> type(32)
>>> type(18.5)
>>> type("Alice")
>>> type(False)
```

Quels sont les quatre réponses obtenus ?

Réponse : On obtient :

```Python
>>> type(32)
<class 'int'>
>>> type(18.5)
<class 'float'>
>>> type("Alice")
<class 'str'>
>>> type(False)
<class 'bool'>
```

On utilise `int` pour représenter un entier relatif, `float` pour un nombre flottant, `str` pour du texte et `bool` pour une information logique.

Le choix d'un type constitue donc une première étape de **modélisation d'une information réelle**.

À retenir

Un programme manipule des représentations d'informations. Le **type** précise la nature des valeurs manipulées et détermine les opérations que Python pourra leur appliquer.

Type	Information représentée	Exemple
`int`	nombre entier relatif	`12`
`float`	nombre décimal approché	`18.5`
`str`	texte	`"Alice"`
`bool`	vrai ou faux	`True`

§2. Écrire des expressions et comprendre leur évaluation

Q2. Prévoir la valeur de chacune des expressions suivantes :

```
>>> 3 + 4
>>> 2 * 5 + 1
>>> "NS" + "I"
>>> -7 / 0
```
Python

Vérifier ensuite vos réponses dans la console. Quel point commun possèdent ces quatre écritures ?

Réponse : Elles produisent respectivement `7`, `11`, `'NSI'` et `ZeroDivisionError`.

Ces quatre écritures sont des **expressions** : Python les évalue pour obtenir une valeur.

À retenir

Une **expression** décrit un calcul permettant d'obtenir une valeur. L'action de déterminer cette valeur est appelée **évaluation de l'expression**.

Q3. On souhaite calculer $3x^2 + 2x + 4$ pour $x = 5$. Tester les expressions suivantes dans la console :

```
>>> 3x^2 + 2x + 4
>>> 3 * 5^2 + 2 * 5 + 4
>>> 3 * 5**2 + 2 * 5 + 4
```
Python

Laquelle de ces expressions permet d'obtenir le résultat attendu ? Quelles différences importantes existe-t-il entre l'écriture mathématique et l'écriture Python ?

Réponse : L'expression correcte est :

```
>>> 3 * 5**2 + 2 * 5 + 4
89
```
Python

Python exige d'écrire explicitement les multiplications avec $*$. La puissance s'écrit $**$ et non ^.

Q4. Prévoir les valeurs des expressions suivantes puis vérifier le résultat de l'évaluation par Python :

```
>>> 7 + 3
>>> 7 - 3
>>> 7 * 3
>>> 7 / 3
>>> 7 // 3
>>> 7 % 3
>>> 7 ** 3
```
Python

Associer ensuite à chaque opérateur son rôle.

Réponse :

`+`	addition
`-`	soustraction
`*`	multiplication
`/`	division
`//`	quotient de la division euclidienne
`%`	reste de la division euclidienne
`**`	puissance

Q5. Déterminer les valeurs des deux expressions suivantes :

```
>>> 17 // 5
>>> 17 % 5
```
Python

Utiliser les résultats obtenus pour compléter l'expression mathématique : $17 = \ldots \times 5 + \ldots$.
Recommencer avec les valeurs 38 et 7. À quoi servent les opérateurs `//` et `%` ?

Réponse : On obtient $17 = 3 \times 5 + 2$ et $38 = 5 \times 7 + 3$.

`//` retourne le quotient de la division euclidienne et `%` son reste.

Q6. Évaluer les expressions suivantes :

```
>>> 8 / 2
>>> 8 // 2
>>> type(8 / 2)
>>> type(8 // 2)
>>> type(8.0 // 2)
```
Python

Que remarquez-vous ?

Réponse : `/` retourne ici `4.0`, de type `float`, tandis que `8 // 2` retourne `4`, de type `int`. Avec `8.0 // 2`, le résultat est un `float`.

§2.1. Dans quel ordre Python effectue-t-il les opérations ?

Q7. Sans utiliser Python, prévoir la valeur des expressions suivantes :

```
>>> 2 + 3 * 4
>>> (2 + 3) * 4
>>> 2 * 3**2
>>> (2 * 3)**2
```
Python

Vérifier ensuite dans la console. Python effectue-t-il simplement les opérations de gauche à droite ?

Réponse : Les résultats sont respectivement `14`, `20`, `18` et `36`. Python applique des règles de priorité ; les parenthèses permettent de modifier l'ordre d'évaluation.

§2.2. Observer pas à pas une évaluation avec Thonny

Dans la partie centrale de l'éditeur Thonny, écrire :

```
2 * (3 + 4)**2 + 5 * 6 // 2
```
Python

Lancer le programme en mode **débogage** puis utiliser la touche F7.

Q8. À l'aide de Thonny, relever l'ordre dans lequel les sous-expressions sont évaluées. Comparer cet ordre avec celui utilisé pour effectuer le calcul à la main.

Réponse : On observe notamment : `3 + 4`, l'élévation au carré, la multiplication par 2, `5 * 6`, la division entière par 2, puis l'addition des deux résultats.

Une expression complexe est évaluée par réduction progressive de sous-expressions plus simples.

§3. Comment utiliser une fonction déjà écrite ?

En mathématiques, une fonction associe à une valeur d'entrée (**l'antécédent**) une valeur de sortie (**l'image**) : pour $f(x) = x^2$, l'entrée 3 donne la sortie 9.

À retenir

En programmation, une **fonction** est un outil qui porte un **nom**, reçoit des données appelées **arguments** et **retourne** un résultat après avoir effectué un traitement. Utiliser une fonction, c'est l'**appeler** en écrivant son nom suivi de parenthèses.

Python fournit déjà de nombreuses fonctions prêtes à l'emploi : on apprend d'abord à s'en servir, avant d'apprendre à en écrire soi-même.

§3.1. De quoi se compose l'appel d'une fonction ?

Q9. Tester les appels de fonctions suivants :

```
>>> abs(-5)
```
Python

```
>>> round(3.14159, 2)
>>> max(4, 9)
```

Pour chaque appel, identifier le nom de la fonction, le ou les arguments et la valeur retournée.

Réponse : Dans `abs(-5)`, `abs` est le nom de la fonction, `-5` l'argument et `5` la valeur retournée. Dans `round(3.14159, 2)`, la fonction `round` reçoit deux arguments et retourne `3.14`.

§3.2. Comment savoir ce que fait une fonction ?

Q10. Évaluer les expressions suivantes :

```
>>> help(abs)
>>> help(round)
```
Python

Quelles opérations réalisent ces fonctions ? Combien d'arguments peut-on leur fournir ? Pourquoi la fonction `help` est-elle importante pour un programmeur ?

Réponse : `help` permet de consulter la documentation d'une fonction. Un programmeur n'a pas à mémoriser toutes les fonctions : il doit savoir consulter leur documentation et comprendre comment les utiliser.

Q11. Évaluer les expressions suivantes :

```
>>> sum(4, 7)
>>> help(sum)
```
Python

Pourquoi le nom d'une fonction ne suffit-il pas pour savoir comment l'utiliser ?

Réponse : L'appel proposé ne correspond pas à l'utilisation attendue de `sum`. La documentation indique notamment que son premier argument doit être un objet itérable, notion qui sera étudiée ultérieurement.

§3.3. À quoi servent les modules ?

De nombreuses fonctions fournies avec le langage Python sont regroupées dans des **modules**.

Q12. Tester le code suivant dans la console :

```
>>> import math
>>> dir(math)
```
Python

Repérer les noms des fonctions qui permettent de calculer : une racine carrée, un sinus, un cosinus et le nom de la variable qui référence la constante π.
Exécuter ensuite `help(math.sqrt)` et calculer la racine carrée de 81.

Réponse : On trouve notamment `math.sqrt`, `math.sin`, `math.cos` et `math.pi`.

```
>>> math.sqrt(81)
9.0
```
Python

Module

Un **module** regroupe des fonctions (et parfois des variables) consacrées à un même domaine : calculs mathématiques, tirages aléatoires, interfaces graphiques, etc. Après :

```python
import math
```

un élément (une fonction par exemple) du module `math` est accessible avec la syntaxe `math.nom`.

Q13. À l'aide de la fonction `help`, découvrir comment et pourquoi utiliser la fonction `random.randint`, puis écrire un appel retournant un entier aléatoire compris entre 1 et 10. Les deux bornes peuvent-elles être obtenues ? Travailler dans la console.

Réponse :

```python
>>> import random
>>> help(random.randint)
>>> random.randint(1, 10)
```

Les deux bornes sont incluses.

§3.4. Que peut-on écrire comme argument d'une fonction ?

Q14. Sans utiliser Python, prévoir ce que retournent ces appels de fonctions, puis les tester dans la console :

```python
>>> abs(3 - 8)
>>> math.sqrt(4 + 5)
>>> abs(3 + 4 / 2 + 7 // 3)
```

Que devient une expression utilisée comme argument d'une fonction ?

Réponse : Python commence par évaluer l'expression utilisée comme argument. Ainsi `abs(3 - 8)` devient d'abord `abs(-5)`, puis produit `5`.

Q15. Dans la partie principale de l'éditeur Thonny, écrire :

```python
import math


math.sqrt(abs(3 - 12))
```

Lancer le programme en mode **débogage** puis utiliser la touche F7. Décrire l'ordre des calculs.

Réponse : Python évalue d'abord $3 - 12 = -9$, puis `abs(-9)` qui vaut 9, puis `math.sqrt(9)` qui vaut 3.

La valeur retournée par une fonction peut donc être utilisée comme argument d'une autre fonction.

Q16. Dans la partie principale de l'éditeur Thonny, écrire :

```python
max(2 + 3, 4 * 2)
```

Lancer le programme en mode **débogage** puis utiliser la touche F7. Relever les valeurs successives prises par chacun des deux arguments avant que la fonction `max` ne soit appliquée. Que peut-on en conclure sur la nature d'un argument ?

Réponse : Thonny évalue d'abord séparément chaque argument : `2 + 3` devient `5` et `4 * 2` devient `8`. L'appel se ramène alors à `max(5, 8)`, qui retourne `8`.

Un argument est une expression : Python la réduit à une valeur avant d'effectuer l'appel de la fonction.

Q17. Prévoir la valeur de chacune des expressions suivantes, puis les écrire dans la partie principale de l'éditeur Thonny :

```python
round(math.sqrt(50), 1)
max(abs(-10), round(7.4))
```

Il sera peut être nécessaire de rechercher à l'aide de la fonction `help` l'action de chacune des fonctions qui doivent être utilisées.

Lancer le programme en mode **débogage** puis utiliser la touche F7. Pour chaque expression, indiquer quelle fonction est appelée en premier, quelle valeur elle retourne, et comment cette valeur est ensuite utilisée.

Réponse : Dans `round(math.sqrt(50), 1)`, Python évalue d'abord `math.sqrt(50)`, qui retourne environ `7.071`. Cette valeur devient le premier argument de `round` : l'appel se ramène à `round(7.071, 1)`, qui retourne `7.1`.

Dans `max(abs(-10), round(7.4))`, les deux arguments sont d'abord évalués : `abs(-10)` retourne `10` et `round(7.4)` retourne `7`. L'appel devient `max(10, 7)`, qui retourne `10`.

La valeur retournée par une fonction peut donc servir d'argument à une autre fonction : c'est la **composition de fonctions**.

À retenir

Appeler (ou utiliser) une fonction, c'est écrire son **nom** suivi de parenthèses contenant ses **arguments**, séparés par des virgules : `round(3.14159, 2)`.

- Une fonction reçoit zéro, un ou plusieurs arguments et **retourne** une valeur après avoir effectué un traitement.
- Chaque argument est une **expression** : Python l'évalue d'abord, puis effectue l'appel.
- La valeur retournée peut être réutilisée, en particulier comme argument d'une autre fonction : `math.sqrt(abs(3 - 12))`.
- Le nom d'une fonction ne suffit pas pour savoir l'utiliser : on consulte sa **documentation** avec `help`.
- Des fonctions supplémentaires sont regroupées dans des **modules**, chargés avec `import` puis utilisés avec la syntaxe `module.nom`.

§4. Comment définir une nouvelle fonction ?

Le langage Python fournit un très grand nombre de fonctions mais il est absolument nécessaire d'être capable d'en définir de nouvelles.

§4.1. Écrire et tester sa première fonction

Q18. Dans l'éditeur, recopier le code suivant :

```python
def carre(x: float) -> float:
    return x**2
```

Puis appeler la fonction depuis la console :

```python
>>> carre(3)
>>> carre(-2)
>>> carre(1.5)
```

Identifier le nom de la fonction, son paramètre, le type de ce paramètre, le type de la valeur retournée, le rôle de `return` dans le corps de la fonction.
Quels sont les arguments utilisés lors de l'appel de la fonction ?

Réponse : La fonction se nomme `carre` et son paramètre est `x`. C'est un `float`. `return` indique la valeur que la fonction retourne ; ici c'est un `float`. Dans `carre(3)`, `3` est un argument.

Paramètres, arguments

Dans la définition :

```python
def carre(x: float) -> float:
    return x**2
```

`x` est un **paramètre**. Dans `carre(5)`, `5` est un **argument**.

Q19. Tester puis corriger la définition de fonction suivante :

```python
def triple(x: float) -> float:
return 3 * x
```

Quelle règle de syntaxe met en évidence l'erreur ?

Réponse :

```python
def triple(x: float) -> float:
    return 3 * x
```

Le corps de la fonction doit être **indenté**.

§4.2. Comment décrire une fonction sans montrer son code ?

Q20. Soit la fonction `surface_rectangle` dont la définition est :

```python
def surface_rectangle(longueur: float, largeur: float) -> float:
    return longueur * largeur
```

Combien possède-t-elle de paramètres ? Quels types sont annoncés pour les paramètres et la valeur retournée ?

Réponse : La fonction possède deux paramètres, `longueur` et `largeur`, annotés avec `float`. La valeur retournée est également annoncée de type `float`.

À retenir

Le **type d'une fonction** est constitué des types de ses paramètres et du type de sa valeur de retour. La **signature** associe le nom de la fonction à son type.

Q21. On donne seulement le type de la fonction `calcul` :

```python
def calcul(x: float) -> float:
    ...
```

Peut-on savoir si cette fonction calcule le carré de x, son double ou son opposé ? Que manque-t-il ?

Réponse : Non. La signature ne suffit pas à déterminer le traitement réalisé. Il manque une description du rôle de la fonction : sa **documentation**.

Q22. Observer la définition de la fonction `carre` :

```python
def carre(x: float) -> float:
    """ Retourne le carré du nombre x. """
    return x**2
```

Quelle partie explique **ce que fait** la fonction ? Quelle partie explique **comment elle le fait** ?

Réponse : La chaîne de documentation explique ce que fait la fonction ; le corps `return x**2` indique comment le calcul est effectué.

À retenir

La **spécification** constitue le contrat entre le concepteur d'une fonction et son utilisateur.

$$\text{Spécification} = \text{Signature} + \text{Documentation}$$

Elle indique essentiellement **ce que fait la fonction, pas comment elle le fait**.

§4.3. Une fonction sans erreur donne-t-elle toujours le bon résultat ?

Q23. On souhaite définir une fonction dont la spécification est :

```python
def retire_un(n: float) -> float:
    """Retourne la valeur de n diminuée de 1."""
```

Un élève propose :

```python
def retire_un(n: float) -> float:
    """Retourne la valeur de n diminuée de 1."""
    return 1 - n
```

Python accepte-t-il cette définition ? Que retourne `retire_un(5)` ? Quelle valeur devrait-elle retourner ? Corriger la fonction.

Réponse : La fonction est syntaxiquement correcte mais `retire_un(5)` retourne `-4` au lieu de `4`.

```python
def retire_un(n: float) -> float:
    """Retourne la valeur de n diminuée de 1."""
    return n - 1
```

§4.4. Peut-on construire une fonction à partir d'une autre ?

Q24. Définir :

```python
def somme(x: float, y: float, z: float) -> float:
    """Retourne la somme de x, y et z."""
    return x + y + z
```

Puis compléter :

```python
def moyenne(a: float, b: float, c: float) -> float:
    """Retourne la moyenne de a, b et c."""
    return ...
```

La fonction `moyenne` doit utiliser la fonction `somme`.

Réponse :

```python
def moyenne(a: float, b: float, c: float) -> float:
    """Retourne la moyenne de a, b et c."""
    return somme(a, b, c) / 3
```

À retenir

Définir une fonction, c'est décrire une fois pour toutes le traitement qu'elle réalise, pour pouvoir ensuite l'**appeler** autant de fois que nécessaire.

```python
def moyenne(a: float, b: float, c: float) -> float:
    """Retourne la moyenne de a, b et c."""
    return (a + b + c) / 3
```

- On commence par le mot-clé `def`, suivi du **nom** de la fonction.
- Entre parenthèses, on liste les **paramètres**, chacun annoté par son **type** (`a: float`). Après `->`, on annonce le **type de la valeur retournée**.
- La ligne d'en-tête se termine par `:`. Le **corps** de la fonction est **indenté**.
- La **documentation** (chaîne entre triples guillemets) décrit **ce que fait** la fonction.
- `return` indique la **valeur retournée** par la fonction, puis met fin à son exécution.
- Le corps d'une fonction peut **appeler d'autres fonctions**, déjà définies ou fournies par Python.

Tant qu'elle n'est pas **appelée** (`moyenne(12, 15, 9)`), une fonction définie n'exécute aucun calcul.

§5. Exercices de synthèse

Consigne : pour chacune des fonctions définies ci-dessous, donner systématiquement leur spécification complète.

§5.1. Exercice 1 : Programmer une fonction polynomiale

Q25. Définir la fonction `polynomiale` qui reçoit les arguments `a, b, c, d, x` et qui retourne le résultat du calcul $ax^3 + bx^2 + cx + d$.
Tests :

```
>>> polynomiale(1, 1, 1, 1, 2)
15
>>> polynomiale(1, 1, 1, 1, 3)
40
```

Réponse :

```
def polynomiale(a: float, b: float, c: float,
                d: float, x: float) -> float:
    """Retourne ax^3 + bx^2 + cx + d."""
    return a * x**3 + b * x**2 + c * x + d
```

§5.2. Exercice 2 : Programmer une moyenne pondérée

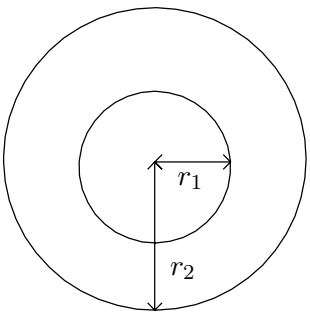
Q26. Définir la fonction `moyenne_ponderee` qui reçoit les arguments `x, y, z, a, b, c` et qui calcule la moyenne pondérée des valeurs x, y, z par les coefficients a, b, c. La fonction doit utiliser la fonction `somme`.

Réponse :

```
def moyenne_ponderee(x: float, y: float, z: float,
                     a: float, b: float, c: float) -> float:
    """Retourne la moyenne pondérée de x, y et z."""
    return (a * x + b * y + c * z) / somme(a, b, c)
```

§5.3. Exercice 3 : Calculer des surfaces et des volumes

Q27. Définir successivement les fonctions `surface_rectangle`, `volume_parallelepipede` en utilisant la fonction précédente, `surface_disque` avec `math.pi`, puis `surface_couronne` avec deux appels à `surface_disque`.



Réponse :

```
def surface_rectangle(longueur: float, largeur: float) -> float:
```

```
    """Retourne la surface d'un rectangle."""
    return longueur * largeur

def volume_parallelepipede(longueur: float,
                           largeur: float,
                           hauteur: float) -> float:
    """Retourne le volume d'un parallélépipède rectangle."""
    return surface_rectangle(longueur, largeur) * hauteur

def surface_disque(r: float) -> float:
    """Retourne la surface d'un disque de rayon r."""
    return math.pi * r**2

def surface_couronne(r1: float, r2: float) -> float:
    """Retourne la surface d'une couronne."""
    return surface_disque(r2) - surface_disque(r1)
```

§5.4. Exercice 4 : Calculer la distance entre deux points

Q28. Dans un repère orthonormé, on peut claculer la distance AB à partir des coordonnées des points A et B :

$$AB = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Définir la fonction `distance` qui reçoit les arguments `x1, y1, x2, y2` et retourne la valeur de la distance entre les deux points.
Tests :

```
>>> distance(0, 0, 3, 4)
5.0
>>> distance(2, 3, 2, 3)
0.0
>>> distance(3, 4, 0, 0)
5.0
```

Réponse :

```
def distance(x1: float, y1: float,
             x2: float, y2: float) -> float:
    """Retourne la distance entre deux points du plan."""
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
```

§5.5. Exercice 5 : Reconstruire un tirage aléatoire

Q29. Consulter la documentation de la fonction `random.randint`.

Définir la fonction `tirage_entier` qui reçoit comme argument un entier `a` et qui retourne un entier aléatoire compris entre 1 et a, bornes incluses. On suppose a entier strictement positif.

Réponse :

```python
def tirage_entier(a: int) -> int:
  """ retourne un entier compris entre 1 et a.
  return random.randint(1, a)
```

Q30. Reprendre l'exercice précédent mais cette fois n'utiliser que la fonction `random.random` du module `random`, la multiplication, l'addition et la fonction `int`. Justifier l'expression avant de programmer.
Cet exercice est beaucoup plus difficile !

Réponse : `random.random()` retourne un nombre x tel que $0 \leq x < 1$. Donc $0 \leq ax < a$ et `int(random.random() * a)` produit un entier de 0 à $a - 1$. Il suffit ensuite d'ajouter 1 pour réaliser le traitement attendu.

```python
def tirage_entier(a: int) -> int:
    """Retourne un entier aléatoire compris entre 1 et a."""
    return int(random.random() * a) + 1
```

§6. Bilan : où en suis-je ?

Pour chaque ligne, cochez la colonne qui correspond. Toute case **Pas encore** indique une partie à reprendre.

Je sais…	Oui	Pas encore	À revoir
choisir un **type** (`int`, `float`, `str`, `bool`) pour représenter une information			Partie 1
prévoir la valeur d'une **expression** et l'ordre dans lequel Python effectue les calculs			Partie 2
appeler une fonction, puis repérer ses **arguments** et sa valeur de retour			Partie 3
trouver comment utiliser une fonction inconnue avec `help`, et importer un **module**			Partie 3
définir une fonction avec `def` et `return`, corps indenté			Partie 4
rédiger la **spécification** d'une fonction et vérifier qu'elle la respecte			Partie 4